

ROS1 vs ROS2

Sergio Arias Hernández y Pablo Cano García

12 de Enero de 2023



Índice

1. Introducion	3
2. Instalación de ROS 2	3
3. Conceptos y como empezar	4
3.1. Conceptos	4
3.2. Primeros pasos	5
3.2.1. Creación de un paquete	6
3.2.2. Publisher y Subscriber	6
4. Diferencias entre ROS1 y ROS2	9
4.1. Protocolo de comunicación	9
4.2. El Nodo Master	10
4.3. Sistemas Operativos	10
4.4. Procesos y Nodos	11
4.5. Librerías	11
4.6. Comandos	11
4.7. Estructura de paquetes	12
4.8. Soporte RTOS	12
5. Conclusión	12
6. Referencias	13

1. Introducion

Robot Operating System (ROS) es una plataforma de software libre y abierta para programación robótica en tiempo real. ROS2 es una evolución de ROS1 que ofrece una mayor escalabilidad, soporte de tiempo real, capacidades de seguridad, entre otras mejoras.

Al igual que ROS1, ROS2 es una plataforma de software modular, escalable y extensible para la programación robótica. Proporciona una amplia variedad de herramientas para ayudar a los desarrolladores a construir aplicaciones de robots avanzadas. Estas herramientas incluyen una amplia gama de bibliotecas, herramientas de simulación, herramientas de programación, herramientas de diseño de arquitectura, herramientas de prueba y herramientas de seguridad.

ROS2 ofrece una variedad de características que permiten a los desarrolladores crear robots más inteligentes, seguros y escalables. Estas características incluyen el soporte para comunicación entre nodos, soporte para lenguajes de programación comunes, soporte para protocolos de comunicación seguros, soporte para una amplia variedad de plataformas, soporte para la creación de aplicaciones de robots, herramientas de simulación y herramientas de prueba.

Está diseñado para usarse con una amplia variedad de robots, desde robots móviles hasta robots industriales y de servicio y permite a los desarrolladores crear robots más inteligentes, seguros y escalables.

ROS2 también ofrece una variedad de servicios para los desarrolladores de robots. Estos servicios incluyen la administración de nodos, la administración de tareas, el intercambio de datos entre nodos, la administración de recursos, la monitorización de la salud de los nodos, la administración de la seguridad de los nodos y el seguimiento y depuración de los nodos.

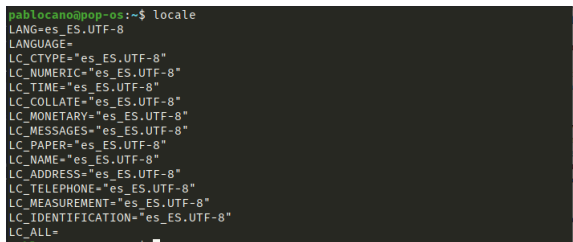
Durante el desarrollo de este trabajo se explica como realizar la instalación de ROS2, los principales conceptos, como empezar a usarlo y las diferencias respecto a ROS1.

https://youtu.be/ncAsJnK_piY

2. Instalación de ROS 2

En este caso se va a tratar la instalación de ROS2 Humble Hawksbill en su versión para Ubuntu. Para ello es necesario tener Ubuntu 22 y realizar los siguientes prerequisites:

1. Poner el formato de localización correcto, ya que ROS2 necesita que esté basado en codificación UTF-8. Para ello se puede hacer uso del comando **locale** y comprobar si termina en UTF-8 tal y como se muestra en la siguiente figura. En caso de que no sea compatible se debera cambiar desde el panel de configuración a una de tipo UTF-8.



```
pablocano@pop-os:~$ locale
LANG=es_ES.UTF-8
LANGUAGE=
LC_CTYPE="es_ES.UTF-8"
LC_NUMERIC="es_ES.UTF-8"
LC_TIME="es_ES.UTF-8"
LC_COLLATE="es_ES.UTF-8"
LC_MONETARY="es_ES.UTF-8"
LC_MESSAGES="es_ES.UTF-8"
LC_PAPER="es_ES.UTF-8"
LC_NAME="es_ES.UTF-8"
LC_ADDRESS="es_ES.UTF-8"
LC_TELEPHONE="es_ES.UTF-8"
LC_MEASUREMENT="es_ES.UTF-8"
LC_IDENTIFICATION="es_ES.UTF-8"
LC_ALL=
```

Figura 1: Ejemplo de uso de locale

2. Activar los repositorios universales de Ubuntu esto se puede hacer empleando los siguiente comandos:

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

3. El siguiente paso es añadir la clave GPG de ROS2 de esta forma:

```
sudo apt update && sudo apt install curl
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
-o /usr/share/keyrings/ros-archive-keyring.gpg
```

4. Por último, se debe añadir el repositorio a lista de fuentes del sistema:

```
echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release &&
echo $UBUNTU_CODENAME) main" | sudo tee
/etc/apt/sources.list.d/ros2.list > /dev/null
```

Una vez se han realizado los prerequisites, se puede proceder con la instalación de ROS2, empleando únicamente los siguientes comandos:

```
sudo apt update
sudo apt upgrade
sudo apt install ros-humble-desktop
```

De esta forma se instalaran todos los paquetes de ROS2 en su totalidad. Una vez hecho esto, se puede inicializar el entorno ROS2 ejecutando el siguiente comando:

```
source /opt/ros/humble/setup.bash
```

Una vez completado estos pasos, ROS2 estará instalado en el sistema y estará listo para usar. Se puede verificar si la instalación fue exitosa ejecutando el siguiente comando:

```
ros2 -v
```

Este comando debería mostrar la versión de ROS2 que acaba de instalar.

3. Conceptos y como empezar

3.1. Conceptos

Antes de comenzar a desarrollar paquetes y aplicaciones con ROS2 es necesario aclarar algunos conceptos:

- **Nodos:** Un nodo es un componente de software que realiza una tarea específica en el sistema. Los nodos se pueden comunicar entre sí a través de mensajes.

- **Mensajes:** Los mensajes son paquetes de información que se envían entre nodos. ROS2 proporciona una amplia variedad de tipos de mensajes predefinidos, como números enteros, flotantes, vectores y matrices, y también permite la creación de mensajes personalizados.
- **Publishers y subscriber:** Los publishers y subscriber son mecanismos de comunicación entre nodos. Un nodo publicador envía mensajes a un topic, mientras que un nodo suscriptor recibe mensajes de ese topic.
- **Topics:** Un topic es un canal de comunicación a través del cual los nodos pueden enviar y recibir mensajes. Los topics se utilizan para permitir que los nodos compartan información y coordinen sus actividades, cada uno está asociado con un tipo de mensaje específico.
- **Servicios:** Los servicios son mecanismos de comunicación entre nodos que permiten solicitar y recibir una respuesta de otro nodo.
- **Paquetes:** Los paquetes son conjuntos de código y recursos relacionados que se utilizan para realizar una tarea específica. ROS2 proporciona una amplia variedad de paquetes predefinidos para realizar tareas comunes en robótica, como el procesamiento de imágenes, la localización y el mapeo.
- **Acciones:** Las acciones son similares en muchos aspectos a los servicios, son un mecanismo de comunicación entre nodos que permite realizar tareas de manera asíncrona. Las acciones se utilizan cuando se necesita realizar una tarea que toma un tiempo indeterminado, como mover un robot a una ubicación específica o realizar una tarea compleja. El servidor de acciones envía periódicamente actualizaciones de estado al solicitante hasta que el objetivo tiene éxito o falla.

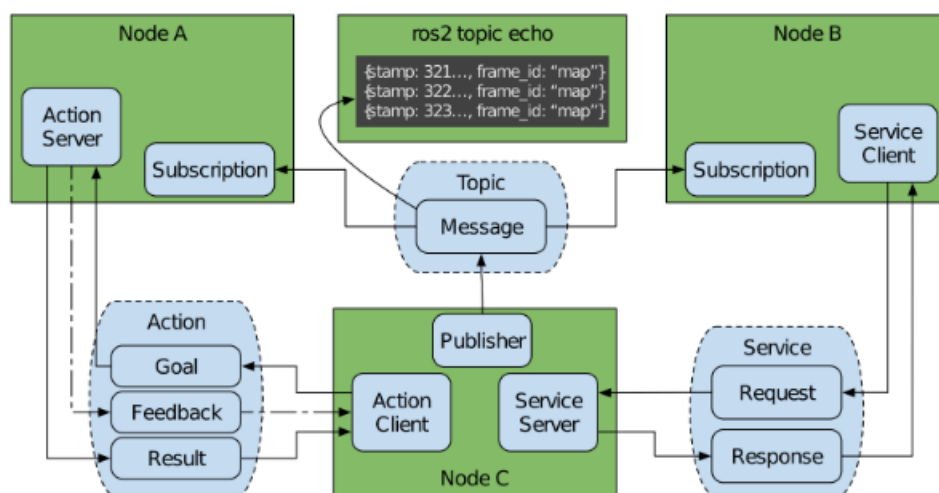


Figura 2: Interfaces de ROS2

3.2. Primeros pasos

En primer lugar, se debe crear un espacio de trabajo para ROS2, para ello se debe crear un directorio como por ejemplo **ros2_ws** con una carpeta **src** dentro del mismo. Dentro de la carpeta **src** será donde se encuentren los distintos paquetes que se quieran emplear. También es importante inicializar ROS2 en la terminales que se utilicen mediante el source.

```
source /opt/ros/humble/setup.bash
```

3.2.1. Creación de un paquete

Para crear un paquete en ROS2 hay que dirigirse a la carpeta `src` del directorio de trabajo y crear en esta un directorio con el nombre del paquete deseado.

Una vez hecho esto, se debe elegir el lenguaje del paquete a crear C++ o Python y se le debe indicar las dependencias necesarias usando el siguiente comando:

```
ros2 pkg create <nombre-paquete> --dependencies [deps] --build-type ament_cmake
```

Crea el paquete en C++.

```
ros2 pkg create <nombre-paquete> --dependencies [deps] --build-type ament_python
```

Crea el paquete en Python.

Este comando creará una serie de archivos y carpetas en el directorio del paquete. Entre estos archivos se encuentra **package.xml** con información sobre el paquete, autor, descripción, etc.

Si se ha creado un paquete en C++ se debe editar el archivo **CMakeLists.txt** y añadir:

```
add_executable(mi_nodo src/mi_nodo.cpp)
```

Siendo **mi_nodo.cpp** el fichero con el código fuente para el paquete, el código fuente se debe situar en la carpeta **src** del paquete.

Si se ha creado un paquete en Python se tendrá un archivo **setup.py** que básicamente equivale al archivo **CMakeLists.txt** pero en Python. En esta archivo se pueden configurar distintos parámetros tales como qué instalar, dónde instalarlo, cómo vincular dependencias, etc.

Por último para compilar los paquetes realizados, hay que situarse en la carpeta raíz del directorio de trabajo `/ros2_ws` y ejecutar lo siguiente:

```
colcon build
```

Una vez que el paquete se haya compilado correctamente, para poder usar el nuevo paquete así como los ejecutables del mismo, en esta misma ruta se debe ejecutar el siguiente comando:

```
. install/local_setup.bash
```

Una vez inicializado, se puede ejecutar el nodo utilizando el siguiente comando:

```
ros2 run mi_paquete mi_nodo
```

3.2.2. Publisher y Subscriber

Tras la creación del paquete anteriormente, se va a proceder a explicar una serie de nodos los cuales van a intercambiar información a través de un topic de forma que un nodo publica información y el otro se suscribe a un topic para recibir esta información.

Para ello se va a seguir el ejemplo disponible en la documentación de ROS2 de la versión Humble, ya que ROS2 dispone de multitud de ejemplos y demos para el aprendizaje. En primer lugar se tiene el código del Publisher:

```
1 import rclpy
2 from rclpy.node import Node
3
4 from std_msgs.msg import String
```

Estas líneas importan las librerías necesarias para el ejemplo. **rclpy** es la biblioteca de cliente ROS2 para Python, y **Node** es una clase de **rclpy** que se usa para crear nodos ROS2. El tipo de mensaje String es un tipo de mensaje predefinido en ROS2.

```
1 class MinimalPublisher(Node):
2
3     def __init__(self):
4         super().__init__('minimal_publisher')
```

Este código define una nueva clase llamada MinimalPublisher, que es una subclase de Node. La función `__init__` se llama cuando se crea una instancia de la clase y se utiliza para inicializar el nodo al que se denomina "minimal_publisher".

```
1         self.publisher_ = self.create_publisher(String, 'topic', 10)
```

Esta línea crea el Publisher que define el tipo de mensaje que se envía, el nombre del topic por donde lo hace y el tamaño de la cola.

```
1         timer_period = 0.5 # seconds
2         self.timer = self.create_timer(timer_period, self.timer_callback)
3         self.i = 0
```

Este código crea un temporizador que llama al método `timer_callback` cada 0.5 segundos especificado por `timer_period`. Además, se inicializa una variable `i` en 0, que se usará en el método `timer_callback`.

```
1     def timer_callback(self):
2         msg = String()
3         msg.data = 'Hello World: %d' % self.i
4         self.publisher_.publish(msg)
5         self.get_logger().info('Publishing: "%s"' % msg.data)
6         self.i += 1
```

El método `timer_callback` se llama cada vez que el temporizador se activa. Crea un nuevo mensaje de String y asigna una cadena que dice "Hello Worldz el valor de `i` definido anteriormente para conocer el número de mensajes que se estan enviando. Luego publica el mensaje en el tópico "topic" utilizando el Publisher creado anteriormente y muestra un mensaje en la consola con el contenido del mensaje que está publicando.

```
1 def main(args=None):
2     rclpy.init(args=args)
3
4     minimal_publisher = MinimalPublisher()
5
6     rclpy.spin(minimal_publisher)
7
8     # Destroy the node explicitly
9     # (optional - otherwise it will be done automatically
10    # when the garbage collector destroys the node object)
11    minimal_publisher.destroy_node()
12    rclpy.shutdown()
13
14 if __name__ == '__main__':
15     main()
```

La función `main` se encarga de inicializar el nodo y hacer que comience a funcionar. Crea una instancia de `MinimalPublisher` y luego llama a `rclpy.spin()` para hacer que el nodo comience a publicar mensajes. Al final, el nodo se destruye y se cierra ROS2.

Una vez se tiene el código del nodo, se deben añadir las dependencias necesarias para el correcto funcionamiento del mismo. Para ello hay que dirigirse al directorio del paquete creado

y modificar el archivo **package.xml** configurando autor, versión, etc y añadiendo las siguiente líneas con las dependencias:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Posteriormente, hay que dirigirse al archivo **setup.py** haciendo coincidir los campos de autor, versión, etc con los especificados anteriormente. Además para añadir el entry point se debe añadir lo siguiente dentro de los corchetes de entry_points quedando de la siguiente forma:

```
1 entry_points={
2     'console_scripts': [
3         'talker = py_pubsub.publisher_member_function:main',
4     ],
5 },
```

De esta forma el nodo publisher ya se encuentra finalizado pudiendo continuar con la elaboración del nodo Subscriber.

Para el nodo Subscriber se tiene el código siguiente:

```
1 import rclpy
2 from rclpy.node import Node
3
4 from std_msgs.msg import String
```

Estas líneas importan lo mismo que en el ejemplo del Publisher.

```
1 class MinimalSubscriber(Node):
2
3     def __init__(self):
4         super().__init__('minimal_subscriber')
5         self.subscription = self.create_subscription(
6             String,
7             'topic',
8             self.listener_callback,
9             10)
10        self.subscription # prevent unused variable warning
```

Este código define una nueva clase llamada MinimalSubscriber, la cual es una subclase de Node. En este caso el nodo se denomina "minimal_subscriber". En este caso, crea una suscripción al topic que permite recibir mensajes de tipo String y con un tamaño de cola de 10. Cuando se recibe un mensaje, se pasa al método listener_callback.

```
1     def listener_callback(self, msg):
2         self.get_logger().info('I heard: "%s"' % msg.data)
```

El método listener_callback se llama cada vez que se recibe un mensaje en la topic y muestra el contenido del mensaje por la consola.

```
1 def main(args=None):
2     rclpy.init(args=args)
3
4     minimal_subscriber = MinimalSubscriber()
5
6     rclpy.spin(minimal_subscriber)
7
8     # Destroy the node explicitly
9     # (optional - otherwise it will be done automatically
10    # when the garbage collector destroys the node object)
11    minimal_subscriber.destroy_node()
12    rclpy.shutdown()
```



```

13
14 if __name__ == '__main__':
15     main()

```

La función `main`, al igual que en el `Publisher`, se encarga de inicializar el nodo y hacer que comience a funcionar. Crea una instancia de `MinimalSubscriber` y luego llama a `rcipy.spin()` para hacer que el nodo comience a procesar mensajes. Al final, el nodo se destruye y se cierra ROS2.

Por último se debe abrir de nuevo el archivo `setup.py` y añadir el entry point del nodo `Subscriber` en el mismo lugar mencionado anteriormente quedando este archivo de la siguiente forma:

```

1 entry_points={
2     'console_scripts': [
3         'talker = py_pubsub.publisher_member_function:main',
4         'listener = py_pubsub.subscriber_member_function:main',
5     ],
6 },

```

4. Diferencias entre ROS1 y ROS2

ROS en su inicio no se creó teniendo en cuenta el uso comercial, por lo que aspectos como la seguridad, la topología de la red y el tiempo de actividad del sistema no se priorizaron. Entonces, con la creciente popularidad de ROS, estas fallas se hicieron cada vez más evidentes.

Es por ello que se desarrolló ROS2, el cual busca tener un uso comercial teniendo en cuenta la seguridad, los sistemas embebidos, las diversas redes y respuestas en tiempo real.

La principal diferencia entre ROS1 y ROS2 se encuentra en la capa intermedia.

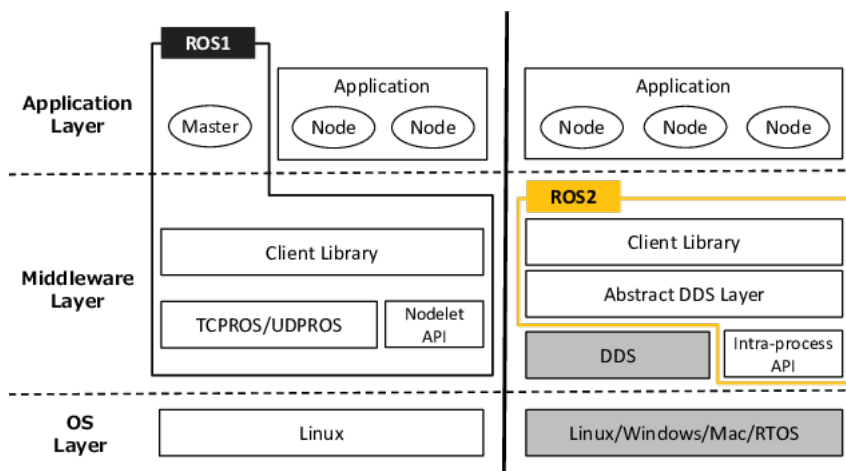


Figura 3: Diferencias en las capas

4.1. Protocolo de comunicación

En ROS2 se utiliza DDS (Data Distribution Service), un protocolo de comunicación estándar para la comunicación entre diferentes usuarios muy utilizado en infraestructuras críticas como acorazados y sistemas espaciales. El uso de DDS brinda garantías de seguridad que no están

presentes en ROS1 y permite la comunicación través de redes inseguras (como Internet).

Mientras, ROS1 está construido con el protocolo TCP/IP el cual se desempeña bien en redes confiables, lo cual puede ser suficiente; sin embargo, en situaciones de redes poco confiables, TCP/IP tiene dificultades para brindar un rendimiento confiable debido a las retransmisiones de datos.

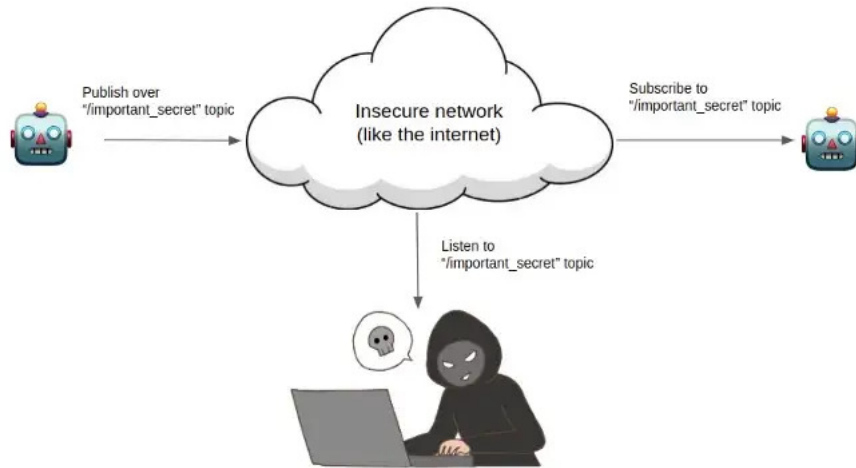


Figura 4: Falta de seguridad en ROS1

4.2. El Nodo Master

En ROS1, existe el nodo ROS Master, que proporciona servicios de registro y nombres para nodos de ROS y actúa como mediador para establecer conexiones entre nodos.

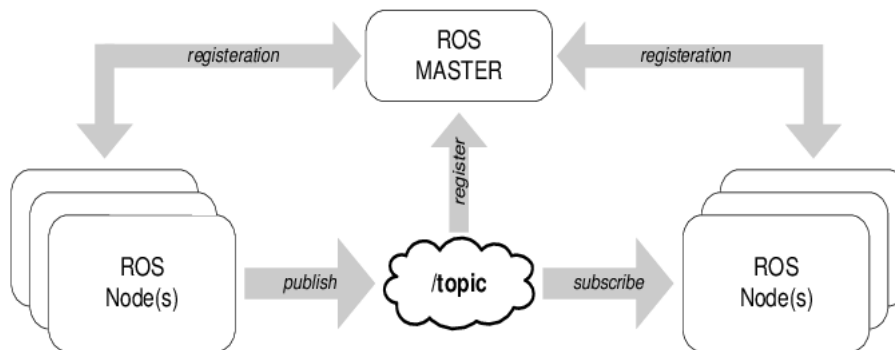


Figura 5: Nodo Master en ROS1

Si en algún momento el nodo Master muere, los nodos existentes se seguirán comunicándose entre sí, sin embargo, si aparece uno nuevo e intenta comunicarse con otro nodo, no tendrá forma de hacerlo porque es el nodo Master el que debe mediar la conexión.

Esto cambia en ROS2, ya que el nodo Master desaparece y toda la comunicación se controla a partir del protocolo DDS sin necesidad de un mediador.

4.3. Sistemas Operativos

Mientras que ROS1 solo es compatible con Linux, ROS2 es compatible con Linux, Windows y macOS y permite integrarse de una manera más sencilla con recursos en la nube como AWS.

4.4. Procesos y Nodos

ROS1 se basa en una arquitectura basada en procesos, mientras que ROS2 utiliza una arquitectura basada en componentes. Esto significa que en ROS1, los nodos son procesos que se comunican a través de mensajes, mientras que en ROS2, los nodos son componentes que se comunican a través de interfaces de publicación y suscripción.

ROS2 cuenta con herramientas para proporcionar información sobre el estado del sistema y de medición integrada de estadísticas de mensajes recibidos por cualquier suscripción, denominada Topic Statistics. Con las Topic Statistics habilitadas, se puede caracterizar el rendimiento del sistema o bien usar los datos para ayudar a diagnosticar cualquier problema presente.

Además en ROS1, los nodos se desarrollaron originalmente para ejecutarse en un solo proceso aunque más tarde se introdujeron los nodelets ROS1, que permiten que varios nodos se ejecuten en el mismo.

ROS2 funciona directamente como los nodelets de ROS1 ejecutando los nodos en el mismo proceso y reduciendo la latencia de comunicación.

4.5. Librerías

ROS2 se basa en librerías estándar como estándar C++11 y los estándares de la industria como DDS. Mientras que ROS1 se basa en librerías propietarias como Boost y OpenCV. Esto hace que en muchos casos sea necesario realizar modificaciones en los códigos para poder ejecutarlo en ROS2.

4.6. Comandos

Respecto a los comandos, la principal diferencia que hay en ROS2 respecto a ROS1 es que, para realizar cualquier comando, siempre empieza con **“ros2”**.

Los principales comandos en ROS2 respecto a ROS1 han cambiado de la siguiente forma:

- **Compilación:** Pasa de *catkin_make* a utilizar la herramienta *colcon*, siendo el comando *colcon build*.
- **Nodos:**
 - “*roslaunch <pkg_name> <node_name>*” pasa a “*ros2 run <pkg_name> <node_name>*”
 - “*roslaunch list*” se convierte en “*ros2 node list*”
 - “*roslaunch info <node>*” pasa a escribirse como “*ros2 node info <node>*”
- **Topics:**
 - “*rostopic list*” pasa a “*ros2 topic list*”
 - “*rostopic info <topic>*” se convierte en “*ros2 topic info <node>*”
- **Servicios:** “*rosservice call <service> [args]*” se convierte en “*ros2 service call <service> [args]*”
- **Launch:** “*roslaunch <pkg_name> <launch_name>*” se convierte en “*ros2 launch <pkg_name> <launch_name>*”
- **Mensajes:** “*rosmmsg show <msg_type>*” pasa a ser “*ros2 interface show <msg_type>*”

Además, en ROS2 se incluye el comando *doctor* el cual verifica todos los aspectos de ROS2, incluida la plataforma, la versión, la red, el entorno, los sistemas en ejecución y más, y advierte sobre posibles errores y motivos de los problemas.

4.7. Estructura de paquetes

Los paquetes en ROS1 admiten varios lenguajes de programación, como C++, Python y Lisp, por el contrario, ROS2 solo admite C++ y Python.

Por otro lado, en ROS2 los archivos roslaunch están escritos en Python, Yaml o XML para admitir una ejecución más configurable y condicionada, por el contrario en ROS1 están escritos en XML reduciendo su capacidades.

Además, según el lenguaje que se utiliza en ROS2 la estructura del paquete es diferente. Los paquetes de ROS2 con Python o con CMake tienen cada uno su propio contenido mínimo requerido. Si se ha compilado para C++ la estructura será la siguiente dentro de `ros2_ws/src/my_package`:

- **package.xml**: archivo que contiene metainformación sobre el paquete.
- **CMakeLists.txt**: archivo que describe cómo construir el código dentro del paquete.
- **src**: Lugar en el que se colocan los scripts escritos en C++.

Mientras que si se ha compilado para Python la estructura dentro de `ros2_ws/src/my_package` es:

- **package.xml**: archivo que contiene metainformación sobre el paquete.
- **setup.py**: contiene instrucciones sobre cómo instalar el paquete.
- **setup.cfg**: se requiere cuando un paquete tiene ejecutables, para que `ros2 run` pueda encontrarlos.
- **/<package_name>**: un directorio con el mismo nombre que el paquete, utilizado por las herramientas de ROS 2 para encontrarlo, contiene `__init__.py` y es donde se colocan todos los scripts escritos en Python.

4.8. Soporte RTOS

ROS2 incorpora un soporte para sistemas embebidos, como la plataforma NVIDIA Jetson, así como soporte para sistemas de tiempo real. Esto significa que ROS2 puede ser utilizado para aplicaciones donde se requiere respuesta en tiempo real. Por el contrario, ROS1 no cuenta con ningún soporte para sistemas embebidos y sólo cuenta con un soporte limitado para sistemas de tiempo real.

5. Conclusión

En conclusión, ROS2 fue diseñado para solucionar los problemas que ROS1 no podía atender, como la seguridad, la topología de la red y los sistemas embebidos. Esto se logra gracias a su protocolo de comunicación DDS, la desaparición del nodo Master, la compatibilidad con

sistemas operativos como Windows, macOS y Linux así como la ejecución más configurable y condicionada. Con todo ello, ROS2 se ha convertido en una herramienta muy útil para realizar diversas tareas robóticas.

Además, presenta mejores características que ROS1, sin embargo, debido a estos cambios y a todo el desarrollo realizado con ROS1, el cambio se esta produciendo de manera muy lenta ya que hay cosas que no son compatibles o hay que rehacerlas para poder utilizar ROS2 donde ahora se utiliza ROS1.

6. Referencias

- <https://maker.pro/ros/tutorial/robot-operating-system-2-ros-2-introduction-and-getting-started>
- [ROS2 Humble Documentation](#)
- [ROS2 Demos](#)
- [ROS2-Command-Cheats](#)
- Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, William Woodall. Robot Operating System 2: Design, architecture, and uses in the wild. American Association for the Advancement of Science, may 2022.